Algoritmia e Desempenho em Redes de Computadores
1.º semestre - 2017/2018

Prefix Trees and Longest Prefix Matching

Rui Figueiredo (78247), Jorge Sacadura (78537), Alexandre Candeias (78599)

I. Introdução

O trabalho descrito consiste na implementação de um conjunto de algoritmos que permitem representar e interagir com uma tabela de prefixos. As tabelas de prefixos são usadas nos *routers* da rede *internet* de modo a definir o caminho de expedição dos pacotes. Para isto o endereço deve ser comparado com as entradas da tabela de prefixos e deve ser escolhido o prefixo mais especifico para aquele dado endereço.

II. Implementação

De modo a reduzir o número de comparações necessárias para encontrar o prefixo mais longo para um determinado endereço propõem-se uma representação da tabela de prefixos como uma árvore binária. A árvore binária é representada em memória por um conjunto de nós. Cada nó possui informação da saída a seguir para um dado prefixo, (*Next-Hop*), que pode ser um número qualquer inteiro e positivo, se a saída não for possível nesse determinado nó o *Next-Hop* tomará o valor correspondente ao ultimo nó válido. Cada nó possui ainda dois ponteiros para duas sub-árvores, a sub-árvore direita e a sub-árvore esquerda, a que corresponde o endereço ter o bit a avaliar a 0 ou a 1.

A. Construção da Árvore Binária

A primeira etapa do programa implementado consiste em ler de um ficheiro a tabela de prefixos, de seguida representa-se em memória esta tabela na forma de uma árvore através da estrutura descrita anteriormente. Inserir um prefixo na árvore binária é uma operação relativamente simples, basta navegar até ao nó correspondente e definir o valor do *Next-Hop* presente na tabela de prefixos. Se o nó correspondente não existir ou se alguns nós no caminho não existirem os mesmos devem ser também criados neste processo.

B. Look Up

A operação de *Look Up* consiste em navegar pela árvore com base no endereço fornecido e chegar o mais para baixo possível, de modo a encontrar o prefixo mais especifico e desta forma o *Next-Hop*. Como podemos chegar a um ponto da árvore em que não existe mais caminho para navegar para um determinado endereço e o nó em que termina a nossa procura não tem saída, precisamos de ir guardando o ultimo *Next-Hop* conhecido ao longo do nosso caminho.

Algorithm 1: LookUp

```
input : RootNode , Address
output: NextHop
begin
    actualNode := RootNode
    lastHop := −1
    for i := 0 to 14 do
        if actualnode → NextHop is valid then
            lastHop := actualNode → NextHop
        end
        if Address[i] is 0 then
            actualNode:=actualNode→ leftChild
        end
        else
            actualNode:=actualNode→ rightChild
        end
        if actualNode not exists then
            break
        end
    end
end
```

C. DeletePrefix

A operação de *DeletePrefix* consiste em eliminar um prefixo da árvore. A operação torna-se um pouco mais complexa pois dependendo da configuração da árvore o número de nós a apagar pode ser mais que um.

São três tipos de casos que podem ocorrer quando se tenta apagar um nó:

- Tipo 1 : O nó a apagar é uma folha e o pai deste é um nó válido ou tem filhos válidos.
- Tipo 2 : O nó a apagar não é uma folha.
- Tipo 3 : O nó a apagar é uma folha e o pai deste não é um nó válido ou não tem filhos válidos.

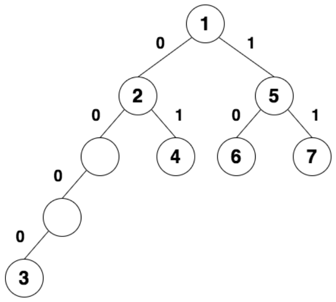

Fig. 1: Exemplos dos vários tipos de configurações possíveis.

Os vários tipos enunciados podem ser encontrados na árvore da figura 1. O tipo 1 corresponde a apagar o nó correspondente ao prefixo 11. O tipo 2 corresponde a apagar o prefixo 0. Já o tipo 3 corresponde a apagar o prefixo 0000.

Inicialmente tentou-se uma solução em que a árvore era percorrida várias vezes. Na primeira iteração era apagado o nó em questão e nas subsequentes era analisados os pais e os avós desse nó para analisar se ainda continuavam nós válidos, ou seja, nós com saída ou com filhos.

Seguidamente chegou-se à conclusão que seria melhor ao percorrer a árvore à procura do nó a apagar ir guardando o nó a partir do qual se pode apagar, baseado nó critério de o nó ser vazio e ter só um filho. O pseudocódigo para o algoritmo implementado é apresentado em 3.

D. PrintTable

A operação de imprimir a árvore binária consiste em fazer uma busca em profundidade na árvore binária de uma forma recursiva. O prefixo que se vai construindo ao percorrer a árvore é acumulado através de um parâmetro da função de recursiva. O prefixo e o correspondente *Hop* é impresso para os nós em que há um *hop* válido.

Algorithm 3: Algoritmo DeletePrefix

```
input  : RootNode , Prefix
output: Void
begin
    len := lenght of prefix
    curr := RootNode
    father := empty
    for i := 0 to len do
        bit := bit i of Prefix
        ant := curr
        if bit is 0 then
            curr := curr → LeftChild
        end
        else
            curr := curr → RightChild
        end
        if curr not exist then
            return ;    ▷ prefix does not
              exist in tree
        end
        if validOutput(ant) or haveTwoChilds(ant)
          then
            father := curr
        end
    end
    if node to delete is a leaf then
        DeleteBinaryTree(father)
    end
    else
        curr → nextHop := −1 ; ▷ no hop for
          this node
    end
end
```

E. BinaryToQuarternaryBit

De modo a diminuir o número de operações necessárias na procura do *NextHop* para um dado endereço, é possível reestruturar a árvore proposta anteriormente de modo a reduzir os níveis da mesma. Para isso agrupámos conjuntos de 2 bits em cada nó da árvore, tendo assim apenas prefixos de tamanho par. Esta abordagem tem alguns problemas principalmente nos prefixos que tem um tamanho ímpar de bits. Para efetuar a conversão entre árvores implementámos o algoritmo apresentado em 4.

O algoritmo 4 tem uma complexidade $\mathcal{O}(n)$, sendo n o número de elementos presentes na árvore binária inicial, o que é justificado por estarmos a varrer toda a

Algorithm 4: CONVERTER ÁRVORE

```
input  : binaryRoot, twobitRoot, address
output: void
begin
    for i := 0 to 2 do
        if (binaryRoot → childs[i] is empty) then
            address = address + 'i';
            Converter
              Árvore(binaryRoot→childs[i],
              twobitRoot, address);
        end
    end
    if binaryRoot not empty then
        if length(address) is even then
            insertPrefix(twobitRoot, address,
              binaryroot→nextHoop);
        else
            if binaryRoot → childs[0] is empty
              then
                insertPrefix(twobitRoot, address +
                  '0', binaryroot→nextHoop);
            end
            if binaryRoot → childs[1] is empty
              then
                insertPrefix(twobitRoot, address +
                  '1', binaryroot→nextHoop);
            end
        end
    end
end
```

árvore com uma *DFS* e a inserir *n* elementos na nova árvore.

III. RESULTADOS

A. Teste 1 - Árvore Enunciado

O teste realizado com a árvore fornecida teve como saída a mesma árvore quaternária que está como exemplo no enunciado.

B. Teste 2 - Ficheiro Exemplo Fénix

O teste realizado com o ficheiro da árvore apresentado no fénix resultou na árvore quaternária apresentada em 2.

C. Teste 3 - Interação

Efetuou-se ainda um terceiro teste interagindo com o programa e realizando as várias operações enunciadas e todas as operações tinham o comportamento previsto na sua implementação.

```
Prefix table Even:
00 6
0010 1
0011 1
01 6
10 5
1000 2
1001 2
1010 3
101000 8
101001 8
1011 7
11 5
1110 4
1111 4
```

Fig. 2: Árvore quaternária para o segundo teste.

IV. CONCLUSÕES

Através dos testes realizados conseguimos validar os algoritmos propostos. Este trabalho permitiu também concluir que a forma como os dados são representados tem uma grande implicação na complexidade dos algoritmos. Se a tabela de dados fosse representada numa lista normal, sem recorrer à representação em árvore, o problema de encontrar o *Hop* para um determinado endereço era no pior dos casos $\mathcal{O}(n)$ em que n é o número de entradas da tabela de prefixos. Com a representação em árvore o algoritmo de procura tem complexidade $\mathcal{O}(l)$ em que l é o número de bits do endereço. A passagem para árvore quaternária reduz a complexidade para $\mathcal{O}(l/2)$.